


C# Avançado

Programação Assíncrona e Paralela




Softblue
cursos online

Tópicos Abordados



- Programação assíncrona e paralela
- Pool de threads
- Delegates assíncronos
 - Exceções
 - Espera pela execução
- Tarefas com a classe *Task*
 - Criando tarefas
 - Exceções
 - Espera pela execução
 - Cancelamento

Tópicos Abordados



- Assincronismo com *await* e *async*
 - Funcionamento
 - Fluxo de execução
 - Tasks com retorno
 - Synchronization context
- Classe *Parallel*
 - *For()*
 - *ForEach()*
 - *Invoke()*
- PLINQ

Programação Assíncrona

• Baseada em métodos que retornam antes de terminarem a execução

Modelo Sincrono

Thread #1 → Execute() → Retorna após o término do método

Modelo Assíncrono

Thread #1 → Execute() → Retorna antes do método terminar

Execute() → Thread #2 → Execute()

Programação Paralela

• Visa dividir um processamento intensivo entre os múltiplos processadores ou núcleos presentes

- Uso de múltiplas threads
- A execução ocorre de forma simultânea

• Dois tipos de paralelismo

- Paralelismo de dados
 - Mesma tarefa executada em múltiplos processadores
 - Cada tarefa atua sobre um subconjunto dos dados
- Paralelismo de tarefas
 - Tarefas diferentes executadas em múltiplos processadores

Assincronismo e Paralelismo

• Programação assíncrona e programação paralela são uma especialização do conceito de programação multithread

Assincronismo

Paralelismo

Programação Multithread

Pool de Threads

• O CLR mantém um pool de threads durante a execução da aplicação

Threads prontas para executarem uma tarefa

Aplicação

O pool promove reutilização das threads

Limita o número de execuções simultâneas

Se houver mais tarefas que threads, é formada uma fila de espera

Considerações Sobre o Pool de Threads

• O número de threads disponíveis pode ser configurado

- Consulte a documentação da classe *ThreadPool*

• Como usar as threads do pool do CLR?

- Delegates assíncronos
- Classe *BackgroundWorker*
- TPL (Task Parallel Library)

• Todas as threads do pool são do tipo *background threads*

Delegates Assíncronos

• Um delegate é capaz de referenciar um ou mais métodos

• A chamada a estes métodos pode ser síncrona ou assíncrona

```
delegate int Operation(int v1, int v2);

static void Main(string[] args)
{
    Operation op = Sum;
    int r = op.Invoke(10, 5);
}

static int Sum(int x, int y)
{
    return x + y;
}
```

Chama o delegate de forma síncrona

A chamada *op(10, 5)* também funciona

Delegates Assíncronos



- Chamadas assíncronas em delegates iniciam com o método *BeginInvoke()*

```
static void Main(string[] args)
{
    Operation op = Sum;
    IAsyncResult ar = op.BeginInvoke(10, 5, null, null);
}
```

- O objeto *IAsyncResult* é uma referência para acessar o resultado do método mais tarde
- O método *EndInvoke()* é chamado

```
int sum = op.EndInvoke(ar);
```

EndInvoke() bloqueia a thread até que o método termine de executar

Exceções em Delegates Assíncronos



- Se uma exceção for lançada pelo delegate, o método *EndInvoke()* vai revelar esta exceção

```
static int Sum(int x, int y)
{
    throw new Exception();
}
```

```
//...
int sum = op.EndInvoke(ar);
```

Neste momento *EndInvoke()* lança a exceção

Mesmo que o delegate não retorne um valor, é importante chamar o *EndInvoke()* para saber a respeito das exceções ocorridas no método

Verificando a Execução do Delegate




- O método *EndInvoke()* bloqueia a thread até que o delegate termine de executar
- É possível também usar a property *IsCompleted* para saber se o delegate já terminou de executar

```
while (!ar.IsCompleted)
{
    Console.WriteLine("Processando...");
}
int sum = op.EndInvoke(handler);
```

EndInvoke() não vai mais bloquear a thread, pois o delegate já terminou

Recebendo Notificação do Delegate



- Ao invés da aplicação ficar testando se o delegate terminou, ela pode ser notificada quando isto acontece


```
IAsyncResult ar = op.BeginInvoke(10, 5, Completed, null);
```

Delegate AsyncCallback

```
static void Completed(IAsyncResult ar)
{
    //...
}
```

Método chamado quando o delegate termina de executar

Recebendo Notificação do Delegate




- É possível passar um parâmetro para o *AsyncCallback* do delegate
 - Tipo *object*

```
Operation op = Sum;
IAsyncResult ar = op.BeginInvoke(10, 5, Completed, op);
```

```
static void Completed(IAsyncResult ar)
{
    Operation op = (Operation)ar.AsyncState;
    int sum = op.EndInvoke(ar);
}
```


A property *AsyncState* acessa o parâmetro

Tarefas com a Classe *Task*



- As classes *Task* e *Task<T>* fazem parte da TPL (Task Parallel Library)
- A TPL contém um conjunto de tipos que permitem o uso da programação paralela
- Task* também é a base para algumas construções da linguagem voltadas à programação assíncrona
- Namespace *System.Threading.Tasks*
- A classe *Task* substitui os delegates assíncronos, trazendo algumas vantagens

Criando Tarefas



- A classe *Task* representa uma tarefa
 - Algum código que deve ser executado por uma thread

```
static void Main(string[] args)
{
    Task.Factory.StartNew(Run);
}

static void Run()
{
    //...
}
```

Inicia uma tarefa usando uma thread do pool

StartNew() retorna um objeto do tipo *Task*


```
Task task = new Task(Run);
task.Start();
```

Separa a criação da execução

```
task.RunSynchronously();
```

Executa a tarefa de forma síncrona

Passando Parâmetros para Tarefas



- É possível fornecer um parâmetro do tipo *object* ao executar a tarefa

```
Task.Factory.StartNew(Run, "AlgunParametro");
```


static void Run(object state)

{

string param = (string)state;

}

Tarefas de Longa Duração



- Quando a tarefa tem por objetivo uma execução durante longo intervalo de tempo, é melhor defini-la como tarefa de longa duração
 - É criada uma nova thread para a tarefa, ao invés de usar uma thread já existente no pool

```
Task.Factory.StartNew(Run, TaskCreationOptions.LongRunning);
```

Consulte a documentação do enum *TaskCreationOptions* para saber a respeito de outras opções de criação de tasks

6

Tarefas que Retornam Valores



- Uma tarefa não precisa obrigatoriamente retornar *void*
- Para tarefas que retornam valores, a classe *Task<T>* é utilizada

```
Task<int> task = Task.Factory.StartNew(() => 5 + 9);
```

A tarefa retorna um *int*

```
Task<int> task = new Task<int>(() => 5 + 9);  
task.Start();
```

Aguardando o Término de uma Tarefa



- Em tarefas que retornam valor, a property *Result* obtém o valor retornado
 - Se a tarefa ainda não terminou, a thread que fez a chamada ao *Result* fica bloqueada

```
Task<int> task = Task.Factory.StartNew(() => Sum(10, 5));  
int sum = task.Result;
```

Obtém retorno do método, bloqueando se ele ainda estiver indisponível

Aguardando o Término de uma Tarefa



- Outra opção para aguardar o término de uma tarefa é o método *Wait()*

```
task.Wait();
```

Retorna *void*

- É possível fornecer um timeout

```
bool finished = task.Wait(1000);
```

Aguarda por no máximo 1s

O retorno indica se a tarefa terminou durante a espera (*true*) ou não (*false*)

Exceções em Tarefas



- Se uma tarefa lançar uma exceção, esta exceção é propagada até o código que está aguardando o término da tarefa
- A exceção propagada é do tipo *AggregateException*, a qual contém a exceção original dentro dela

Exceções em Tarefas



```
void Run()  
{  
    throw new Exception("Erro");  
}
```

```
try  
{  
    task.Result;  
} catch (AggregateException e)  
{  
    Exception originalExc = e.InnerException;  
    //...  
}
```

Exceção propagada como uma *AggregateException*

A property *InnerException* retorna a exceção original

Cancelamento de Tarefas



- Um token de cancelamento pode ser usado para cancelar tarefas em andamento

```
CancellationTokenSource cts = new CancellationTokenSource();  
CancellationToken token = cts.Token;  
  
Task t = Task.Factory.StartNew(() =>  
{  
    while (true)  
    {  
        token.ThrowIfCancellationRequested();  
        Thread.Sleep(2000);  
    }  
}, token);
```

O token é fornecido à tarefa

Se houve cancelamento, lança uma *OperationCanceledException*

A property *IsCancellationRequested* indica se houve cancelamento, sem lançar a exceção

Cancelamento de Tarefas



- O cancelamento é feito através do método *Cancel()*

```
cts.Cancel();
```

- A exceção é propagada como uma *AggregateException*

```
try
{
    t.Wait();
}
catch (AggregateException e)
{
    var oce = (OperationCanceledException)e.InnerException;
    //...
}
```

Assincronismo com *async* e *await*

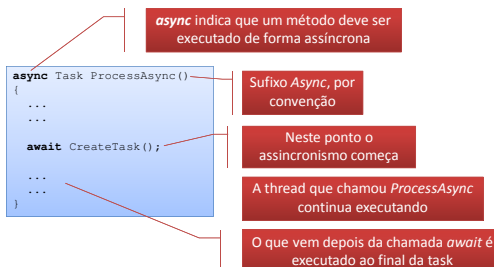


- O uso das palavras-chave *async* e *await* eleva a programação assíncrona a outro nível
- O desenvolvedor apenas indica onde a execução será assíncrona e o CLR cuida do resto
- *async* e *await* trabalham em conjunto
 - *async*
 - Define que um método poderá ser executado de forma assíncrona
 - *await*
 - Bloqueia a thread até que a tarefa seja terminada
 - O bloqueio ocorre em uma nova thread

Exemplo de Uso de *async* e *await*



```
Task CreateTask()
{
    return Task.Run(() => Thread.Sleep(3000));
}
```



Detalhes do *await*



- O *await* só pode ser usado em métodos definidos como *async*
- O *await* deve operar apenas com os tipos *Task* ou *Task<T>*

```
await CreateTask();
```

```
await Task.Run(() => Thread.Sleep(3000));
```

- Um método *async* pode definir mais de um *await*

```
await Task1();  
...  
await Task2();  
...  
await Task3();  
...
```

Cada tarefa será executada após a tarefa anterior terminar

Detalhes do *async*



- Métodos definidos com *async* devem ter pelo menos uma chamada *await*
 - Se não tiver um *await*, o método será síncrono
 - O compilador emite um aviso na falta do *await*
- Métodos *async* só podem ter 3 tipos de retorno
 - *Task*, *Task<T>* ou *void*
- O retorno *void* é usado no tratamento de eventos (em WPF, por exemplo)

Tasks com Retorno



- Um método assíncrono que possui retorno deve retornar um tipo *Task<T>*

```
async Task<int> SumAsync(int x, int y)  
{  
    return await Task<int>.Run(() => x + y);  
}
```

Expressão lambda que retorna um *int*

```
int r = SumAsync(5, 10).Result;
```

Result lê o resultado

Tasks com Retorno



- O retorno pode ocorrer também fora da task

```
async Task<bool> ProcessAsync()  
{  
    await Task.Run(() => Thread.Sleep(3000));  
    return true;  
}
```

A task não retorna informação

O retorno ocorre depois do final da task

O dado retornado deve ser compatível com o tipo de retorno determinado na assinatura do método

Fluxo de Execução



```
ProcessAsync();  
***  
async Task ProcessAsync()  
{  
    for (int i = 1; i <= 5; i++)  
    {  
        Console.WriteLine(i);  
    }  
  
    await Task.Run(() =>  
    {  
        for (char c = 'A'; c <= 'E'; c++)  
        {  
            Console.WriteLine(c);  
        }  
    });  
  
    for (int i = 6; i <= 10; i++)  
    {  
        Console.WriteLine(i);  
    }  
}
```

A thread executa até o disparo da task. Depois ela deixa o método

Uma thread é designada para executar a task

Quando a task termina, outra thread continua a execução do método

Qual thread?

Synchronization Context



- A thread que vai executar o código que vem depois do término da task depende do *synchronization context* configurado
- Aplicações console não tem um contexto definido por padrão
 - Uma thread do pool é designada para execução
- Aplicações WPF usam o contexto da UI thread
 - É a UI thread que executa este código
 - Isto é importante porque componentes da interface gráfica só podem ser manipulados pela UI thread
- Consulte a documentação do C# para mais informações sobre como definir um *synchronization context* customizado

Como o *await* Funciona Internamente



```
void ProcessAsync()
{
    for (int i = 1; i <= 5; i++)
    {
        Console.WriteLine(i);
    }

    Task t1 = Task.Run(() =>
    {
        for (char c = 'A'; c <= 'E'; c++)
        {
            Console.WriteLine(c);
        }
    });

    t1.ContinueWith(t =>
    {
        for (int i = 6; i <= 10; i++)
        {
            Console.WriteLine(i);
        }
    });
}
```

O método *t1.ContinueWith()* executa um código após o término de *t1*

A thread que vai executar este código depende do *synchronization context*

A Classe *Parallel*



- A classe *Parallel* possui métodos estáticos para executar algumas operações em paralelo
- Exemplos
 - *For()*
 - *ForEach()*
 - *Invoke()*

Parallel.For()



- Executa uma estrutura *for*, onde as iterações podem ser executadas de forma paralela

```
for (int i = 0; i < 10; i++)
{
    Console.WriteLine(i);
}
```

Execução sequencial

```
Parallel.For(0, 10, i =>
{
    Console.WriteLine(i);
});
```

Execução paralela

A ordem de execução das iterações não é bem definida

Parallel.For()

• O loop pode ser parado usando o método *Break()*

```
Parallel.For(0, 10, (i, status) =>
{
    if (i == 5)
    {
        status.Break();
    }
    Console.WriteLine(i);
});
```

Tipo *ParallelLoopState*

Não permite a execução das iterações seguintes

O método *Stop()* funciona de forma semelhante, mas todas as threads param após terminar a iteração

Útil quando o algo foi encontrado e as outras iterações não importam

Parallel.ForEach()

• Executa uma estrutura *foreach*, onde as iterações podem ser executadas de forma paralela

```
foreach (string e in array)
{
    Console.WriteLine(e);
}
```

Execução sequencial

```
Parallel.ForEach(array, e =>
{
    Console.WriteLine(e);
});
```

Execução paralela

A ordem de execução das iterações não é bem definida

Parallel.ForEach()

• O índice da iteração também pode ser recuperado

```
Parallel.ForEach(array, (e, state, i) =>
{
    if (i == 2)
    {
        state.Break();
    }
    Console.WriteLine(e);
});
```

Tipo *ParallelLoopState*

Índice

Não permite a execução das iterações seguintes

O método *Stop()* também pode ser usado aqui

Parallel.Invoke()



- Permite executar tarefas diferentes em paralelo
- Um array de delegates é fornecido ao método *Invoke()*

```
Parallel.Invoke(Method1, Method2);
```

```
Parallel.Invoke(  
    () => { Console.WriteLine("A"); },  
    () => { Console.WriteLine("B"); }  
);
```

O método bloqueia até que todas as tarefas tenham terminado

PLINQ



- A linguagem LINQ pode ser usada na extração de dados
- PLINQ é o LINQ paralelo
 - Os dados são extraídos através do uso de múltiplas threads
- PLINQ só funciona com LINQ para objetos
 - Não funciona no LINQ para XML ou no Entity Framework

O Método *AsParallel()*



- O método *AsParallel()* é usado para indicar que o paralelismo deve existir

```
var q = from n in numbers.AsParallel()  
        where n % 2 == 0  
        select n;
```

- O PLINQ divide a execução em partes para que a execução paralela aconteça
- No final, o PLINQ agrupa as execuções
- Todo o processo é transparente ao desenvolvedor

Considerações Sobre o PLINQ



- Nem sempre o paralelismo vai ocorrer
 - O PLINQ pode identificar que a execução sequencial tem performance melhor
- A ordem de execução dos operadores de uma query LINQ deixa de ser respeitada quando o PLINQ é usado
 - É possível que a ordem seja respeitada através do uso do método *AsOrdered()*
 - Este método pode causar queda de performance em coleções muito grandes